Université de Toulouse | **Projet MIDL 1** | Année 2025 - 2026

1 Contexte

Dans l'UE "Philosophie des sciences" en L1, vous avez étudié la procédure d'élimination de quantificateurs dans la théorie des ordres denses (DO). Cette procédure est rappelée en annexe A.

Le but de ce projet est d'implanter cette procédure et d'en étudier (et si possible, implanter) une extension qui est un fragment de l'arithmétique des nombres rationnels.

L'objectif d'apprentissage est, au plan pratique :

— de se familiariser avec la représentation de formules dans un langage comme Python ;
— d'apprendre à parcourir des formules et de les transformer ;
— de choisir une représentation informatique appropriée d'un problème, au-delà de ce qui s'impose à première vue ;
— de faire un usage raisonné d'outils d'Intelligence Artificielle lors de la programmation ;
— de s'initier à de bonnes pratiques de développement de logiciel.

Au plan théorique, il s'agit d'étendre un algorithme donné et d'étudier et justifier ses propriétés.

2 Structure du projet

Le projet est structuré en trois lots dont le premier est obligatoire et les deux autres sont optionnels. Les lots sont en plus décomposés en tâches dont les résultats doivent être rendus au cours du semestre.

Les lots et tâches sont :

— **L1 : Implantation de la procédure d'élimination de la théorie DO** *(obligatoire)*
Votre fonction prend comme entrée une formule du langage de DO, par exemple $F_1 = \forall x.\forall y.\forall z.\ x < y \wedge x < z \longrightarrow y < z$, et elle décide si la formule est vraie (c'est le cas ici) ou fausse.
  — **T1** prise en main des structures de données et fonctions de base (§ 5.1.1)
  — **T2** implantation de la procédure de décision (§ 5.1.2)
  — **T3** finition : tests, rapport et soutenance (§ 5.1.3)
— **L2 : Extension de DO à l'arithmétique des rationnels** *(facultative)*
Tandis que la théorie DO de base admet uniquement la comparaison de deux variables (comme $x < z$), nous nous plaçons ici dans la théorie des ordres pour l'arithmétique des nombres rationnels. Une formule dans cette théorie est $F_2 = \forall x.\exists y.\exists z.\ 3 * x - z < y \wedge z < 5 * x + y$.
  — **T4** Rapport : modifications de la procédure de base pour l'arithmétique (§ 5.2.1)
  — **T5** Implantation de la procédure pour l'arithmétique (§ 5.2.2)
— **L3 : Preuve constructive dans DO + arithmétique des rationnels** *(très facultative)*
Avoir un résultat vrai / faux n'est parfois pas très convainquant pour un être humain. Il vous est demandé ici de justifier la correction d'une formule. Libre cours est laissé à votre imagination, mais voici une piste : vous pouvez créer un *jeux* qui est capable de répondre a tout défi de montrer la correction d'une formule valide (est-ce que F_2 est vrai pour $x = 2$? oui, pour $y = -1$ et $z = 8$) et de réfuter une formule invalide.
  — **T6** Rapport : version constructive pour l'arithmétique (§ 5.3.1)
  — **T7** Implantation de la version constructive (§ 5.3.2)

Notation Une excellente réalisation du lot 1 rapporte 14 points ; chacun des lots 2 et 3 rapporte jusqu'à 4 points supplémentaires (note globale limitée à 20 évidemment).

3 Déroulement du projet

Dates

Le progrès des travaux du lot 1, obligatoire, doit être documenté par des rendus aux dates suivantes :

- **T1 :** le lundi 3 novembre 2025 à 23h (après les vacances de Toussaint)
- **T2 :** le lundi 26 janvier 2026 à 23h (vers la fin de l'inter-semestre)
- **T3 :** le lundi 9 février 20026 à 23h (début du semestre 2)

Une **soutenance** aura lieu début février 2026.

Personne ne vous empêche de rendre vos travaux avant les dates limites.

Pour la réalisation des lots 2 et 3, vous pouvez vous organiser à votre convenance et déposer vos résultats avec le rendu final de la tâche 3.

4 Éléments à rendre

Le travail s'effectue en groupes de 2-3 étudiants (un travail individuel est aussi possible, mais n'est pas conseillé).

Chaque groupe soumet ses ses travaux sur la page Moodle prévue à cet effet.

Tâche	Type	Date limite	Remarque
T1	Code	3 novembre 2025	obligatoire
T2	Code	20 janvier 2026	obligatoire
T3	Rapport et Code	9 février 2026	obligatoire
T4	Rapport		facultatif, avec T3
T5	Code		facultatif, avec T3
T6	Rapport		facultatif, avec T3
T7	Code		facultatif, avec T3

Le **code** doit

- ou bien être déposé sur Moodle dans une archive (`tarfile`) avant la date limite. Merci d'utiliser uniquement tar [1] et surtout pas de formats propriétaires.
- ou bien nous parvenir dans un mail avec un lien vers un dépôt Git ou Mercurial que nous pouvons cloner. Le mail doit être envoyé avant la date limite. Nous nous engageons a ne pas récupérer le dépôt avant la date limite.

Une exigence minimale est que le code est bien formé et peut être exécuté. Il est souhaitable qu'il soit correctement documenté. La documentation de haut niveau va dans le rapport, la documentation des fonctions individuelles dans le code. Les deux devraient être complémentaires et pas redondants.

Le **rapport** doit être déposé directement sur Moodle, en format PDF, de préférence écrit en LaTeX pour ne pas faire subir au lecteur le formatage cruel de formules en Word et Cie.

- Le rapport pour le lot 1 doit comporter :
  - une courte *introduction* qui décrit le problème sans entrer dans les détails techniques, et qui résume le travail effectué, et peut-être aussi ce que vous n'avez pas réussi à faire ($\frac{1}{3}$ ou $\frac{1}{2}$ page) ;
  - une petit *manuel d'utilisateur* qui indique comment utiliser votre code ;
  - une description de haut niveau de l'implantation.
- Les rapports pour les lots 2 et 3 sont des sections supplémentaires du rapport pour le lot 1 (un seul document).

Notez que la réflexion théorique des tâches 4 et 6 est un préalable des implantations dans les tâches 5 et 7. Par contre, il est tout à fait possible de se limiter aux tâches 4 et 6 ; un argumentaire claire et solide sera valorisé aussi sans implantation.

1. https://www.gnu.org/software/tar/manual/tar.html

5 Indications sur le travail à effectuer

5.1 Procédure pour la théorie DO

5.1.1 Structures de données et fonctions de base

L'implantation se fait en Python. Les structures de données vous seront fournies dans un fichier `syntax.py`. Nous vous conseillons de les utiliser telles quelles sans les modifier, mais vous avez le droit de le faire et devez alors l'indiquer dans votre documentation.

La représentation d'une formule est une instance d'une *classe* de formules, une notion provenant de la programmation orientée objet dont vous pouvez entièrement vous passer pour effectuer le travail demandé.

Construction de formules Le niveau d'abstraction qu'il convient d'adopter est qu'une **formule** est construite à partir de constructeurs, selon le principe que vous connaissez du cours Logique 1 (Définition inductive des formules). Une formule est inductivement générée par les constructeurs suivants :

- une constante. Le constructeur `ConstF` prend un argument Booléen : `ConstF(False)` pour $\bot$ et `ConstF(True)` pour $\top$
- une comparaison entre deux variables. Le constructeur `ComparF` prend trois arguments : le nom de la première variable (un string), un opérateur de comparaison, et le nom de la deuxième variable. Un opérateur de comparaison est ou bien `Eq` (pour $=$) ou bien `Lt` (pour $<$). Avec cela, la formule $x = y$ se construit avec `ComparF("x", Eq(), "y")` et $y < z$ avec `ComparF("y", Lt(), "z")`.
- une négation d'une formule f. Le constructeur `NotF` prend un argument, la formule f.
- un opérateur booléen entre deux formules f et g. Le constructeur `BoolOpF` prend trois arguments : les formules et l'opérateur, `Conj` ou `Disj`. Ainsi, $f \wedge g$ s'écrit `BoolOpF(`f`, Conj(), `g`)`. Parce que c'est lourd à écrire, nous avons introduit les fonctions `conj` et `disj` qui permettent d'écrire directement, par exemple, `conj(`f`,`g`)`. Cette apparente complexité a ses avantages parce qu'elle permet de réduire le nombre de constructeurs et de profiter de la dualité des opérateurs $\wedge$ et $\vee$.
- la quantification sur une formule. Le constructeur `QuantifF` prend trois arguments : un quantificateur (`All` ou `Ex`), le nom de la variable, et une formule.

La représentation de variables liées par un nommage explicite des variables n'est pas la plus astucieuse [2] mais la plus facile à comprendre.

> *Note :* Pour éviter certaines manipulations fastidieuses, nous nous limitons à des formules en forme prénexe (tous les quantificateurs au début de la formule).

Exemple : Souvenez-vous de la définition de confluence de la relation $<$, dont la forme prénexe est : $\forall x.\forall y.\forall z.\ \exists u.\ x < y \wedge x < z \longrightarrow y < u \wedge z < u$

Après conversion de $\longrightarrow$, sa représentation est :

```
QuantifF(All(), 'x',
 QuantifF(All(), 'y',
    QuantifF(All(), 'z',
       QuantifF(Ex(), 'u',
         BoolOpF(NotF(BoolOpF(ComparF('x', Lt(), 'y'), Conj(), ComparF('x', Lt(), 'z'))),
                 Disj(),
                 BoolOpF(ComparF('y', Lt(), 'u'), Conj(), ComparF('z', Lt(), 'u')))))))
```

2. voir les indices de de Bruijn comme alternative, https://fr.wikipedia.org/wiki/Lambda-calcul#Les_indices_de_de_Bruijn

Utilisez les abréviations, y compris pour $\longrightarrow$ pour une écriture un peu plus lisible :

```
allq("x",
  allq("y",
    allq("z",
      exq("u",
       impl(conj(ltf("x", "y"), ltf("x", "z")),
            conj(ltf("y", "u"), ltf("z", "u")) )))))
```

Notez que la représentation interne est toutefois la même qu'avec les constructeurs de base.

L'affichage de la formule `f` définie comme en haut se fait en Python avec :

```
>>> print(f)
∀x.(∀y.(∀z.(∃u.((¬(((x < y) ∧ (x < z))) ∨ ((y < u) ∧ (z < u)))))))
```

Décomposition de formules On peut décomposer une formule par des *accesseurs* qui sont données dans les définitions de classe. Prenons la définition des formules quantifiées :

```
class QuantifF(Formula):
    q: Quantif
    var: str
    body: Formula
    def __str__(self):
        return f"{self.q}{self.var}.({self.body})"
```

Cette définition dit que le constructeur `QuantifF` a trois arguments, comme déjà mentionné. Elle dit aussi que vous pouvez extraire le quantificateur avec l'accesseur `q`, la variable avec `var` et la formule avec `body`. L'application des accesseurs peut être itérée. Pour la formule `f` de l'exemple :

```
>>> f.q
All()
>>> f.var
'x'
>>> f.body.body.body.body.right.left
ComparF(left='y', op=Lt(), right='u')
```

La définition `def __str__(self) ...` remplace l'affichage brut des constructeurs lors d'un `print` par une version plus lisible.

Manipulation de formules Comme vous savez du cours Logique 1, les formules sont transformées le plus souvent en parcourant l'arbre syntaxique récursivement jusqu'aux feuilles.

Très souvent, vous pouvez appliquer le schéma suivant, illustré par une fonction (plutôt inutile) qui convertit les $\wedge$ d'une formule en $\vee$ et inversement.

```
def dualOp (op: BoolOp) -> BoolOp:
    if isinstance(op, Conj):
        return(Disj())
    else:
        return(Conj())

def dual(f: Formula) -> Formula:
    if isinstance(f, ConstF):
        return f
    elif isinstance(f, ComparF):
```

```
        return f
    elif isinstance(f, NotF):
        return NotF(dual(f.sub))
    elif isinstance(f, BoolOpF):
        return BoolOpF(dual(f.left), dualOp(f.op), dual(f.right))
    else:
        raise ValueError("dual␣applied␣to␣quantified␣formula")
```

La fonction auxiliaire `dualOp` convertit les opérateurs. La fonction récursive `dual` sur les formules teste successivement à quelle classe appartient l'instance actuelle, et applique le traitement approprié. Ici, sur les constantes et comparaisons, elle ne fait rien (c'est à dire, renvoie la formule), se plonge récursivement dans la négation et transforme les opérateurs booléens comme voulu. Un `raise` lève une exception, c'est à dire, arrête l'exécution du programme immédiatement.

```
>>> f = conj(ConstF(True), ConstF(False))
>>> print(f)
(⊤ ∧ ⊥)
>>> dual(f)
BoolOpF(left=ConstF(val=True), op=Disj(), right=ConstF(val=False))
>>> print(dual(f))
(⊤ ∨ ⊥)
>>> dual(QuantifF(All(), "v", ConstF(True)))
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 11, in dual
ValueError: dual applied to quantified formula
```

> *Note :* Pour éviter des effets de bord, n'essayez pas de *modifier* une formule existante, du style `f.left = dual(f.left)`, mais construisez une nouvelle formule avec le constructeur, comme dans `BoolOpF(dual(f.left), ...)`

5.1.2 Implantation de la procédure de décision

Un rappel de la procédure se trouve en annexe A.2. Elle semble complexe mais se compose facilement en fonctions qui ne sont pas difficiles.

Les descriptions en § A.2.2 et § A.2.3 font croire que nous manipulons *une* formule. L'enjeu essentiel est de trouver des représentations et structures de données qui facilitent la manipulation. Voici quelques pistes :

- En § A.2.2, est-ce qu'un transforme en une formule en forme normale disjonctive, ou est-ce qu'il y a de meilleures représentations ? Pensez à un ensemble de clauses vu en Logique 1.
- Aussi "l'élimination des quantificateurs à partir de l'intérieur"ne se fait pas sur une formule monolithique. Pensez à décomposer une formule de la forme $\exists x.\exists y.\exists z.\phi$ en une liste $[x, y, z]$ (dans cet ordre ?) et la formule ϕ. En fait, il faut aussi prendre en compte d'autres informations, mais traitez d'abord des cas simples pour généraliser après.
- Enfin, la suppression d'une variable (en § A.2.3) ne se fait pas sur une formule. Si vous avez déjà un format clausal, une représentation s'impose.

> *Note :* Ce sont ces aspects de représentation que vous devrez mettre en exergue dans votre rapport écrit, peut-être discuter des représentations alternatives et motiver vos choix.

Lisez aussi les remarques sur le langage Python en annexe B.

5.1.3 Finition

Cette dernière partie vous permet :

— de prendre en compte nos remarques pour les tâches T1 et T2 pour améliorer la qualité et lisibilité de votre code (commentaires, typage etc.)
— de rajouter des tests
— de produire une trace de l'exécution de la procédure ou des explications, peut-être avec différents niveaux de détail qui peuvent être configurés par l'utilisateur de votre code.

Le produit final doit être du code Python qui tourne dans une REPL (Read-Eval-Print Loop) de Python. Il n'est pas demandé de concevoir une interface plus conviviale. Toute autre interface se heurterait à la difficulté principale qui est de saisir la formule qu'on souhaite simplifier. Il suffit si les formules sont écrites comme applications de constructeurs comme en § 5.1.1 ; n'essayez pas d'écrire un parser pour des formules.

5.2 Extension de DO à l'arithmétique des rationnels

Note préalable : Vos enseignants ont développé un prototype pour le travail décrit en § 5.1, mais ce n'est plus le cas pour ce qui suit. Il est possible que l'élimination de quantificateurs échoue pour l'extension proposée ici. Ce qui est valorisé est la justesse de votre argumentation et non pas le succès ou échec de la procédure.

5.2.1 Arithmétique (théorie)

Tandis que la théorie DO de § 5.1 manipule des termes de base de la forme $x < y$ ou $x = y$, l'extension permet de comparer deux expressions e_1, e_2, de la forme $e_1 < e_2$ ou $e_1 = e_2$. Les expressions e_i sont des termes linéaires en plusieurs variables avec des coefficients rationnels (écrits comme des flottants ou entiers), par exemple $3 * x - z$ ou $5.42 * x + y$.

Votre travail consiste à scruter précisément la procédure de annexe A et de dire à quels endroits elle doit être modifiée. L'idée essentielle de la procédure pour *DO* restera la même : on élimine successivement les variables d'une formule, en utilisant des propriétés comme la transitivité et la densité pour remplacer, par exemple, $y < x < z$ par $y < z$, pour variables y, z (étape (5) de annexe A.2.3) - sauf que les inégalités sont entre expressions e_1, e_2 et il faut isoler la variable x : $e_1 < x < e_2$. Élaborez les détails.

5.2.2 Arithmétique (implantation)

Sur la base de vos réflexions théoriques, vous pouvez implanter la procédure de décision étendue. S'il y a raison de le faire, vous pouvez adapter ou généraliser des fonctions de base (par exemple calcul des formes normales négatives) au lieu de les dupliquer.

Ici aussi, l'enjeu essentiel est de trouver des structures de données appropriées :

— modifiez la structure de données des formules pour permettre la comparaison d'expression au lieu de variables.
— réfléchissez à une bonne représentation d'expressions de l'arithmétique linéaire à plusieurs variables. Vous aurez peut-être envie de choisir une définition qui permet de représenter tout arbre syntaxique avec des additions et multiplications, comme $2 * (x + 3 * y)$ ou $(x + y) * (2 + z)$, mais vous en découvrez vite les inconvénients (il serait souhaitable de les commenter dans votre rapport). Vous avez le droit d'imposer une structure de données qui est suffisamment "normalisée" pour empêcher la représentation d'expressions illégales et pour faciliter des calculs (isolation d'une variable).

Si vous voulez aller beaucoup plus loin, vous pourrez explorer différentes méthodes de l'élimination de Fourier-Motzkin [3] et la théorie des corps réels clos [4].

3. https://fr.wikipedia.org/wiki/%C3%89limination_de_Fourier-Motzkin
4. https://fr.wikipedia.org/wiki/Corps_r%C3%A9el_clos#Th%C3%A9orie_des_corps_r%C3%A9els_clos

5.3 Preuve constructive

5.3.1 Preuves constructives (théorie)

Ce travail essaie d'explorer le rapport entre logique et la théorie des jeux :

— Pour montrer la *validité* d'une formule, un joueur (qui représente les quantificateurs $\exists$) essaie de montrer la validité en proposant des instances des variables universelles qui rendent la formule vraie. L'opposant, qui représente les quantificateurs $\forall$, essaie de réfuter la formule donnée en proposant des instances des variables existentielles qui rendent la formule fausse. Si le joueur $\exists$ a une stratégie gagnante, la formule est valide. Si le joueur $\forall$ a une stratégie gagnante, la formule n'est pas valide parce qu'il peut la réfuter.

— Pour une formule *insatisfiables*, les rôles de $\exists$ et $\forall$ sont inversés.

Par exemple, la formule $\exists x.\forall y.\exists z.x = 42 \vee x < y \wedge y < z$, est valide parce que le joueur $\exists$ a une stratégie gagnante : il peut choisir $x = 42$. Quel que soit le choix pour y, le joueur $\forall$ n'a plus d'opportunité de rendre la formule fausse. L'exemple illustre aussi que le choix des variables n'est pas aléatoire, un mauvais choix peut mener à une impasse : après le choix $x = 43$, le joueur $\forall$ peut toujours faire un choix (par exemple $y = x - 1$) qui invalide la formule.

Le jeu s'effectue entre l'ordinateur et un joueur humain. L'ordinateur analyse une formule donnée et élabore une stratégie de preuve (respectivement de réfutation) gagnante si la preuve est valide (respectivement insatisfiable).

Réfléchissez à une méthode de génération d'une telle stratégie, qui est effectivement un algorithme inspiré par la manière dont les variables sont éliminées.

5.3.2 Preuves constructives (implantation)

Implantez ce jeux, dont le travail essentiel réside encore une fois dans la bonne représentation de la stratégie.

A Procedure d'élimination de quantificateurs

Ceci est un résumé des notions que vous trouvez aussi sur les transparents du cours "Philosophie des sciences (MIDL)" du semestre 2 de L1.

A.1 *DO* : théorie des ordres denses sans bornes

Elle est axiomatisée pour une relation $\prec$ avec les propriétés :

— (Trans) : $\forall x, y, z.\ x \prec y \wedge y \prec z \longrightarrow x \prec z$ [transitive]

— (Asym) : $\forall x, y.\ x \prec y \longrightarrow \neg(y \prec x)$ [asymétrique]

— (Conn) : $\forall x, y.\ x = y \vee x \prec y \vee y \prec x$ [connectivité]

— (Dense) : $\forall x, y.\ x \prec y \longrightarrow \exists z.\ x \prec z \wedge z \prec y$ [ordre dense]

— (SExtr) $\forall x.\ \exists y, z.\ y \prec x \wedge x \prec z$ [sans extrema]

Par facilité d'écriture, nous utilisons souvent $<$ au lieu de $\prec$.

A.2 Résumé de la procédure

A.2.1 Hypothèses

La procédure prend une formule F et la convertit en $\top$ ou $\bot$ par élimination de quantificateurs

Exigences :

— F doit être close (sans variables libres)

— Sinon : fermer avec $\forall x_0 \dots x_n.\ F$ pour $\{x_0 \dots x_n\} = fv(F)$

— Les seules symboles relationels permis sont : ordre $\prec$, égalité $=$; pas de fonctions ou constantes.

Démarche :

1. Convertir F en forme prénexe
2. Convertir quantif. universels en existentiels : $(\forall x.\phi) \leftrightarrow \neg(\exists x.\ \neg\phi)$
   Attention : ce n'est pas une forme normale négative
3. Éliminer les quantificateurs à partir de l'intérieur

> *Remarques :* Vous pouvez supposer que la formule est déjà close et en forme prénexe. Il ne reste donc que les étapes (2) et (3). Bien sûr, les extensions décrites en § 5.2 introduisent des fonctions et constantes, mais vous pouvez les ignorer dans un premier temps.

A.2.2 Prétraitement

Simplification d'une formule de la forme $\exists x.\ \phi$

1. Tirer les négations à l'intérieur : forme normale négative
2. Éliminer les négations devant les relations :
   (a) $\neg(z \prec z') \leftrightarrow (z = z' \vee z' \prec z)$
   (b) $\neg(z = z') \leftrightarrow (z \prec z' \vee z' \prec z)$
3. Transformer en forme normale disjonctive : $\phi \leftrightarrow \bigvee_j \psi_j$
4. Tirer la quantification existentielle à l'intérieur de la disjonction :
   $\exists x.\ \phi \leftrightarrow \bigvee_j (\exists x.\ \psi_j)$

Les quantifications existentielles intérieures ont maintenant le format $\exists x.\ \psi$, où ψ est une conjonction de relations $u \prec v$ ou $u = v$

A.2.3 Suppression de variable

Suppression de la quantification existentielle de $\exists x.\ \psi$:

1. Si $x \notin fv(\psi)$: $(\exists x.\ \psi) \leftrightarrow \psi$
2. Si ψ contient le terme $x < x$: $(\exists x.\ \psi) \leftrightarrow \bot$
3. Sinon, regrouper les termes de ψ en : $(\bigwedge_i x \prec u_i) \wedge (\bigwedge_j v_j \prec x) \wedge (\bigwedge_k w_k = x) \wedge \chi$ avec $x \notin fv(\chi)$
4. Si $(\bigwedge_k w_k = x)$ présent : choisir une variable w_0 parmi les w_k, et
   $(\exists x.\ \psi) \leftrightarrow (\bigwedge_i w_0 \prec u_i) \wedge (\bigwedge_j v_j \prec w_0) \wedge (\bigwedge_k w_k = w_0) \wedge \chi$
5. Sinon, si $(\bigwedge_i x \prec u_i)$ et $(\bigwedge_j v_j \prec x)$ présents dans ψ :
   $(\exists x.\ \psi) \leftrightarrow \bigwedge_{i,j} v_j \prec u_i \wedge \chi$
6. Sinon, si uniquement $(\bigwedge_i x \prec u_i)$ ou $(\bigwedge_j v_j \prec x)$ présent dans ψ :
   $(\exists x.\ \psi) \leftrightarrow \chi$

Post-traitement : Simplifier conjonctions/disjonctions

B Python

B.1 Version typée de Python

Nous vous recommandons fortement d'utiliser une version typée de Python, où les paramètres d'une fonction et sa sortie sont annotés avec des types. Ces annotations sont une bonne documentation. Des warnings peuvent éliminer certaines erreurs de codage statiquement (avant exécution) et ainsi réduire drastiquement la phase de tests.

Une fonction qui s'écrit

```
def foo(x1, x2):
  ...
```

s'ecrit :

```
def foo(x1 : T1, x2 : T2) -> TR:
  ...
```

avec les annotations, ou les `T1, T2` sont les types des variables et `TR` le type de résultat. Les types peuvent être des types de base : `bool, int, str`, ou aussi `Formula` pour notre type de formules, ou formés à partir de constructeurs de types, par exemple `tuple[T1, T2]` pour des paires de valeurs de types `T1, T2` ou `list[T]` pour une liste d'éléments de type `T`.

Une version banale de la division d'Euclide s'écrit donc :

```
def divEucl(x: int, y: int) -> tuple[int, int]:
    return (x // y, x % y)
```

En présence de sous-types, les messages d'erreurs / warning (affichées par exemple par des éditeurs comme VSCode) ne sont pas toujours correcte (perte d'information).

B.2 Importation et génération de code

Vous pouvez utiliser librement des fonctions de haut niveau disponibles dans la librairie de Python [5]. Vous pouvez aussi utiliser des outils d'intelligence générative pour générer des bouts de code. Dans ce dernier cas, merci d'indiquer dans des commentaires quel outil vous avez utilisé, peut-être en affichant avec quel prompt vous avez obtenu la réponse.

Voici quelques schémas qui sont utiles surtout pour le traitement de listes, qui s'apparentent à des quantifications sur des ensembles ou de la compréhension d'ensembles :

— Compréhension de listes qui construit $\{f(x)|x \in L \wedge P(x)\}$.
Le résultat est une liste : `[f(x) for x in L if P(x)]`
Exemple : tripler les éléments pairs d'une liste :

```
>>> [3 * x for x in [1, 2, 3, 4, 5] if x % 2 == 0]
[6, 12]
```

L'itération se fait aussi sur plusieurs listes. Utile pour former le produit cartésien de deux listes (et ensuite filtrer certains éléments) :

```
>>> [(x, y) for x in [1, 3, 5] for y in [2, 4, 6] if x < y]
[(1, 2), (1, 4), (1, 6), (3, 4), (3, 6), (5, 6)]
```

— Quantification existentielle : $\exists x.x \in L \wedge P(x)$.
Le résultat est un booléen : `any(P(x) for x in L)`.
Exemple : est-ce que la liste contient des nombres pairs ?

```
>>> any (x % 3 == 0 for x in [1, 2, 3, 4, 5])
True
>>> any (x % 3 == 0 for x in [1, 2, 4, 5])
False
```

Similaire : quantificateur universel `all`.

5. https://docs.python.org/3/reference/index.html